HICFD – Hocheffiziente Implementierung von CFD-Codes für HPC-Many-Core-Architekturen

Thomas Alrutz[4], Petra Aumann[7], Achim Basermann[3], Kim Feldhoff[5], Thomas Gerhold[1], Jörg Hunger[8], Jens Jägersküpper[1], Hans-Peter Kersken[3], Olaf Knobloch[7], Norbert Kroll[1], Olaf Krzikalla[5], Edmund Kügeler[2], Ralph Müller-Pfefferkorn[5], Mathias Puetz[6], Andreas Schreiber[3], Christian Simmendinger[4], Christian Voigt[2] und Carsten Zscherp[8]

[1] Deutsches Zentrum für Luft- und Raumfahrt e.V., Institut für Aerodynamik und Strömungstechnik, C²A²S²E: Center of Computer Applications in Aerospace Science and Engineering, Bunsenstr. 10, D-37073 Göttingen, {thomas.gerhold,jens.jaegerskuepper,norbert.kroll}@dlr.de
[2] Deutsches Zentrum für Luft- und Raumfahrt e. V., Institut für Antriebstechnik, Linder Höhe, D-51147 Köln, {edmund.kuegeler,christian.voigt}@dlr.de
[3] Deutsches Zentrum für Luft- und Raumfahrt e. V., Simulations- und Softwaretechnik, Linder Höhe, D-51147 Köln, {achim.basermann,hans-peter.kersken,andreas.schreiber}@dlr.de
[4] T-Systems Solutions for Research GmbH, Bunsenstr. 10, D-37073 Göttingen, {thomas.alrutz,christian.simmendinger}@t-systems-sfr.com
[5] Technische Universität Dresden, Zentrum für Informationsdienste und Hochleistungsrechnen, Zellescher Weg 12, D-01062 Dresden, {kim.feldhoff,olaf.krzikalla,ralph.mueller-pfefferkorn}@tu-dresden.de
[6] IBM Deutschland GmbH, Hechtsheimer Str. 2, D-55131 Mainz, mathias.puetz@de.ibm.com
[7] Airbus Deutschland GmbH, Aerodynamic Tools and Simulation, {petra.aumann,olaf.knobloch}@airbus.com
[8] MTU Aero Engines GmbH, Dachauer Str. 665, D-80995 München, {joerg.hunger,carsten.zscherp}@mtu.de

Zusammenfassung. Bei dem Forschungsprojekt HICFD handelt es sich um ein Verbundprojekt des vom Bundesministerium für Bildung und Forschung geförderten Programms „IKT 2020 – Forschung und Innovation“. Das Forschungsprojekt hat zum Ziel, neue Methoden und Werkzeuge zur Analyse und Optimierung des Leistungsvermögens strömungsmechanischer, paralleler Programme auf Hochleistungsrechnern mit Prozessoren mit einer Vielzahl von Kernen zu entwickeln und diese exemplarisch auf die strömungsmechanischen Codes TAU und TRACE des Projektpartners DLR anzuwenden. Die wesentlichen Ziele des Projektes werden in dieser Arbeit vorgestellt.

Das Leistungsvermögen strömungsmechanischer, paralleler Programme soll in diesem Projekt durch eine optimale Ausnutzung aller Parallelitätsebenen verbessert werden. Auf der obersten Ebene (MPI) soll eine intelligente Gitteraufteilung den Lastausgleich zwischen den MPI-Prozessen verbessern. Für blockstrukturierte Gitter soll hier ein Multi-Core-kompatibles Partitionierungswerkzeug entwickelt werden. Auf der Ebene der Multi-Core-Architektur sollen exemplarisch für die beiden Strömungslöser TAU und TRACE hochskalierende hybride OpenMP/MPI-Verfahren implementiert werden. Auf Prozessorkern-Ebene soll ein Präprozessor entwickelt werden, der die komfortable Nutzung paralleler SIMD-Einheiten („Single Instruction Multiple Data“) auch für komplexe Anwendungen ermöglicht. Um zu überprüfen, in welchem Umfang die Anwendung des Präprozessors das Leistungsvermögen paralleler Programme beeinflusst, soll die Leistungsanalyse-Software Vampir in Richtung SIMD erweitert werden.

Schlüsselwörter: Performance-Analyse, Vampir-Suite, SIMD, Präprozessor, Code-Transformation

1 Einleitung

Der wachsende internationale Konkurrenzdruck stellt die europäische Luftfahrtindustrie vor große Herausforderungen. Als Antwort auf die verschärften Umweltschutzanforderungen strebt sie in der Agenda 2020 an, den durch den Luftverkehr verursachten Lärm zu halbieren, sowie den Kohlendioxid-Ausstoß (und damit den Kerosinverbrauch) um 50 Prozent und die Produktion von Stickoxiden um 80 Prozent zu reduzieren. Gleichzeitig soll die Verkürzung der Entwicklungszeiten um 50 Prozent die Konkurrenzfähigkeit signifikant verbessern. Diese Ziele lassen sich nur durch konsequente Weiterentwicklung der Simulationstechnologie erreichen. Sie erfordern insbesondere den Übergang von der Simulation einzelner physikalischer Phänomene für Einzelkomponenten zu einer multidisziplinären Simulation kompletter Systeme unter realen und instationären Einsatzbedingungen. Die dazu erforderlichen Entwicklungen erfolgen in Europa, und insbesondere in Deutschland, in enger Partnerschaft

von Industrie und Großforschung.

Eine wichtige Rolle spielen hierbei große Simulationscodes, die in öffentlichen Forschungsinstituten unter Verwendung innovativer Algorithmen entwickelt werden und dadurch einen Vorsprung gegenüber kommerziell verfügbaren Produkten aufweisen. Neben dem bei der französischen ONERA entwickelten blockstrukturierten Strömungslöser elsA sind die im DLR entwickelten Codes TAU (Berechnung von Außenströmungen) und TRACE (Reagierende Innenströmungen) die tragenden Säulen der Simulationstechnik in der europäischen Luftfahrtindustrie. Die initiierte Kopplung der Codes TAU und TRACE wird in Zukunft die Simulation noch komplexerer (und damit realistischerer) Strömungsszenarien erlauben. Auf der Hardwareseite genügen für Auslegungsrechnungen i. d. R. mittelgroße Rechnerplattformen mit mehreren Hundert Rechenknoten, während Proof-of-Concept-Studien den Einsatz der jeweils stärksten installierten Rechnerplattformen (beispielsweise die Systeme der Gauß-Allianz) erfordern. Alle diese Systeme basieren bereits heute auf Multi-Core-Technologie. Für die nahe Zukunft ist der Wechsel zu Many-Core-Architekturen absehbar.

2 SIMD-Erweiterung Vampir-Suite

Um herauszufinden, in welchen Teilen die strömungsmechanischen Programme in Richtung SIMD verbessert werden können, soll die Vampir-Suite in diese Richtung erweitert werden. Bei der Vampir-Suite handelt es sich um eine Leistungsanalyse-Software, die Ereignisse (z. B. Funktionsaufrufe) von parallelen Programmen zunächst aufzeichnet (*Tracing*) und diese dann durch *globale* (für alle Prozesse) und *lokale Anzeigen* (für einen bestimmten Prozess) über die Zeit grafisch darstellt (*Visualisierung*) [1]. Abb. 1 zeigt eine typische grafische Anzeige der Vampir-Suite. Im Detail sollen die Tracing- und Visualisierung-Fähigkeiten der Vampir-Suite dahingehend verbessert werden, dass SIMD-Operationen wie z. B. Vektoradditionen oder -multiplikationen über neue grafische Anzeigen besser beobachtet werden können.

Bevor Informationen über die Verwendung von *SIMD-Maschinenbefehlen* (Maschinenbefehlen, die mehrere Daten gleichzeitig verarbeiten können) grafisch dargestellt werden können, müssen diese Informationen in einer geeigneten Form während eines Programmlaufs aufgezeichnet werden. Moderne Mikroprozessoren besitzen spezielle Register, in denen Informationen über verschiedene Ereignisse (z. B. die Anzahl der Gleitkommazahloperationen oder die Anzahl der Cache-Misses) erfasst werden können, auf vielen Prozessoren auch die Anzahl bestimmter SIMD-Maschinenbefehle.

Definition 1 (Hardware-Counter [2]). *Register auf einem Mikroprozessor, die speziell dafür bereit stehen, die Häufigkeiten des Auftretens bestimmter Hardware-Ereignisse oder Hardware-Informationen zu speichern, heißen auch* Hardware-Counter.

Definition 2 (Spezielle Hardware-Ereignisse [2]). *Ein Hardware-Ereignis, das sich (nur) mit Hilfe von Hardware-Countern des zugrundeliegenden Mikroprozessors aufzeichnen lässt, heißt* natives Hardware-Ereignis. *Ein natives Hardware-Ereignis, das sich aufgrund der Verwendung einer bestimmten Bibliothek unter einem weiterem Namen ansprechen lässt, heißt* (von der Bibliothek) voreingestelltes Hardware-Ereignis. *Ein Hardware-Ereignis, das aus anderen Hardware-Ereignissen zusammengesetzt ist, heißt* abgeleitetes Hardware-Ereignis.

Die Tabellen 1, 2 und 3 enthalten Beispiele von nativen, voreingestellten und abgeleiteten Hardware-Ereignissen für zwei verschiedene Prozessoren.

Tabelle 1. SIMD-relevante, native Hardware-Ereignisse, die auf Mikroprozessoren des Typs AMD® Opteron® X85 verfügbar sind.

Nativer Hardware-Ereignisname	Beschreibung
`FR_X86_INS`	Anzahl aller Maschinenbefehle
`FR_FPU_MMX_3D`	Anzahl SIMD-Befehle der Erweiterungen MMX™ und 3DNow!™
`FR_FPU_SSE_SSE2_PACKED`	Anzahl gepackter SIMD-Befehle der Erweiterungen SSE und SSE 2
`FR_FPU_SSE_SSE2_SCALAR`	Anzahl skalarer SIMD-Befehle der Erweiterungen SSE und SSE 2
`FR_FPU_X87_SSE_SSE2_SCALAR`	Anzahl x87-Maschinenbefehle und skalarer SIMD-Befehle (SSE, SSE 2)
`FR_FPU_X87_SSE_SSE2_SCALAR_PACKED`	Anzahl x87-Maschinenbefehle und aller SIMD-Befehle (SSE, SSE 2)

Tabelle 2. Hardware-Counter-spezifische Merkmale von Mikroprozessoren des Typs AMD® Opteron® X85 sowie des Mikroprozessors Intel® Itanium® 2 Montecito 9040 bei Verwendung der Bibliothek PAPI in der Version 3.5.0.

Merkmal	Opteron® X85	Itanium® 2 Montecito 9040
Anzahl Hardware-Counter	4	12
Native Hardware-Ereignisse	248	637
Von PAPI voreingestellte Ereignisse	103	103
davon verfügbar	42	60
davon verfügbar und abgeleitet	13	42

Um in Programmierhochsprachen auf Informationen aus Hardware-Countern zugreifen zu können, werden üblicherweise spezielle Bibliotheken verwendet, die solche Informationen in Form einer API zur Verfügung stellen, Das Tracing-Werkzeug der Vampir-Suite unterstützt die Verwendung der Bibliothek *PAPI* („Performance Application Programming Interface“) [3], [4]. Diese stellt Hardware-Informationen in C und C++ zur Verfügung und enthält in der Version 3.5.0 ca. 100 voreingestellte Hardware-Ereignisse. Damit ist der Zugriff auf Hardware-Counter aus der Vampir-Suite heraus möglich, und es können dann z. B. für verschiedene numerische Benchmarks SIMD-relevante Hardware-Ereignisse aufgezeichnet und visualisiert werden. Ein Beispiel für solche Visualisierungen ist in Abb. 2 gegeben.

3 Skalierbare Algorithmen für Many-Core-Architekturen

In diesem Teilarbeitspaket soll untersucht werden, wie sich die parallele Effizienz von Strömungslösern auf Multiblocknetzen weiter erhöhen lässt. Die Entwicklung eines generischen Many-Core-fähigen Partitionierungswerkzeugs für blockstrukturierte Gitter ist hier unerlässlich. Das Ziel ist zum einen die Entwicklung effektiver Verfahren zur Verteilung sowie zum Splitten und Mergen von Teilblöcken zu einem optimalen Verbund in Abhängigkeit der zur Verfügung stehenden Prozessoren.

Dabei sollen zusätzlich Anforderungen bezüglich Lastbalancierung, Cache-Nutzung, Kaskadierung des Kommunikationsnetzwerks auf parallelen Many-Core-Systemen sowie physikalische und softwaretechnische Nebenbedingungen berücksichtigt werden. Der Partitionierer soll nach Ende des Projekts frei verfügbar gemacht werden.

Folgendes Vorgehen ist geplant:

Tabelle 3. SIMD-relevante, von der Bibliothek PAPI (Version 3.5.0) voreingestellte Hardware-Ereignisse, die auf Mikroprozessoren des Typs AMD® Opteron® X85 verfügbar sind.

Voreingestellter Ereignisname	Nativer Hardware-Ereignisname
`PAPI_FP_INS`	`FR_FPU_X87_SSE_SSE2_SCALAR_PACKED`
`PAPI_VEC_INS`	`FR_FPU_SSE_SSE2_PACKED`
`PAPI_TOT_INS`	`FR_X86_INS`

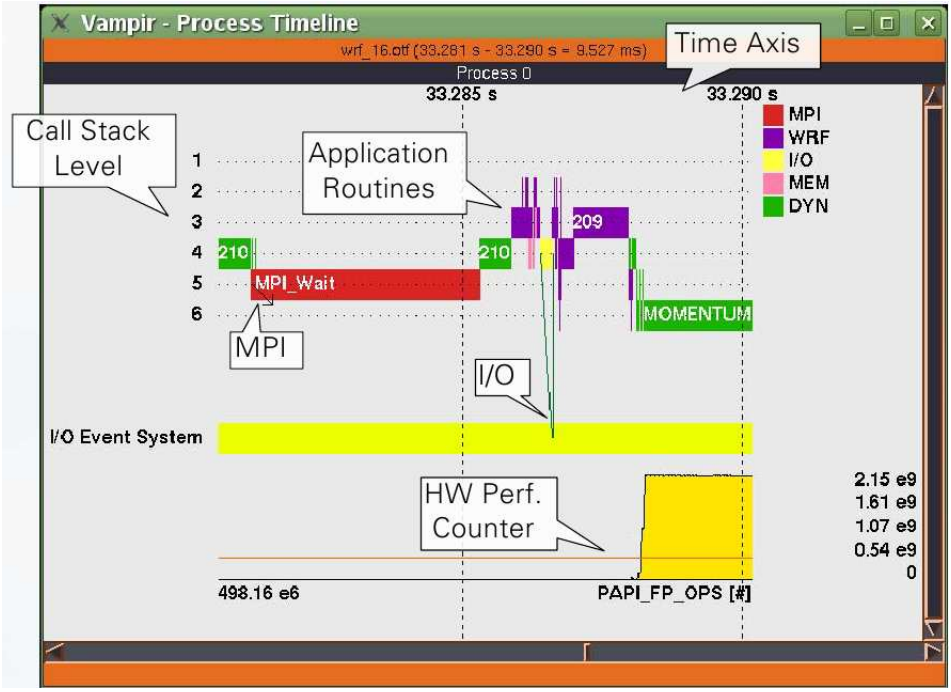

Abb. 1. Typische Anzeige innerhalb der Leistungsanalyse-Werkzeuges Vampir zur Untersuchung des Leistungsvermögens eines parallelen Programms.

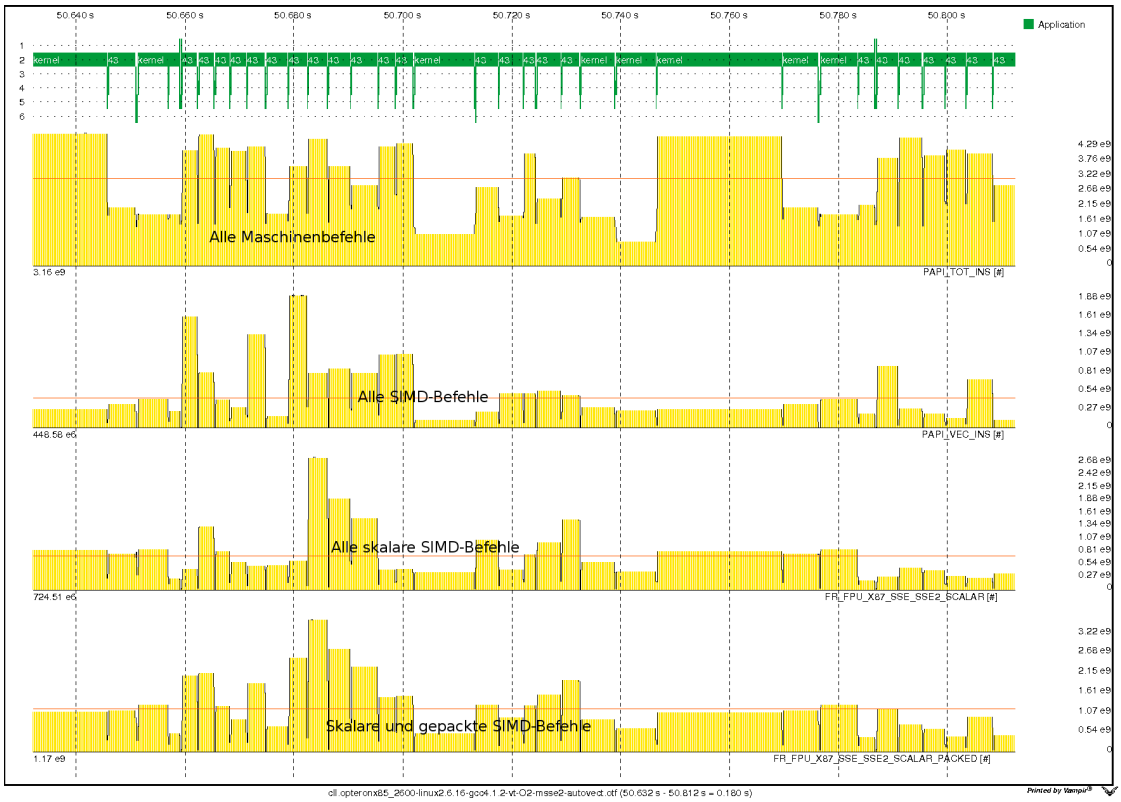

Abb. 2. Grafische Darstellung verschiedener Hardware-Counter über den Programmverlauf eines numerischen Benchmarks.

– Zunächst wird eine Kostenfunktion für die Rechenkosten pro Block definiert. Diese berücksichtigt unter anderem Löseroperationen, insbesondere Vorkonditionierung und Rechnungen bezüglich Randbedingungen, z.B. periodischen Randbedingungen.
– Gemäß der Kostenfunktion sehr teure Blöcke werden dann gesplittet, um später eine lastbalancierte Verteilung der Blöcke auf die einzelnen (Multi-Core-)Prozessoren erreichen zu können.
– Anschließend wird ein „Block"-Graph erstellt. Die Knotengewichte dieses Graphen entsprechen den Rechenkosten pro Block, die Kantengewichte den Kommunikationskosten zwischen den Blöcken.
– Die „Block"-Graphpartitionierung auf die einzelnen Prozessoren kann dann z.B. mit ParMETIS [5] durchgeführt werden.
– Während sich die Graphpartitionierung auf die MPI-Parallelisierung bezieht, können die Berechnungen pro Multi-Core-Prozessor per OpenMP-Parallelisierung weiter auf die einzelnen Cores verteilt werden. Die Datenaufteilung muss hier die Speicherhierarchie des Prozessors berücksichtigen.

Zum anderen muss der Löser für lineare Gleichungssysteme für Multi-Core-Architekturen optimiert und die parallele Skalierbarkeit des Lösungsalgorithmus verbessert werden. Dazu sollen verschiedene Vorkonditionierer in TRACE implementiert und auf ihre Performance untersucht werden.

In einem ersten Test wurde die in *linear-/adjointTRACE* vorhandene Block-Jacobi-Vorkonditionierung (BJ) [6] mit der „Distributed Schur Complement"-Methode (DSC) [7] anhand eines typischen Matrix-Problems verglichen. Die BJ-Methode berücksichtigt lediglich die lokalen Diagonalblöcke der verteilten Matrix und vernachlässigt Kopplungen zwischen den Teilgebieten. Daher verschlechtert sich der Vorkonditionierungseffekt mit steigender Prozessorzahl. Andererseits nimmt der Aufwand zur Berechnung des Vorkonditionierers ab. Die DSC-Methode hingegen berücksichtigt alle Kopplungen zwischen den Teilgebieten. Der Aufwand zur Bestimmung und Anwendung des Vorkonditionierers nimmt jedoch mit wachsender Prozessorzahl in der Regel zu, da die Zahl der Kopplungen in der Regel ansteigt.

Abbildung 3 vergleicht die Ausführungszeiten eines parallelen FGMRes-Lösers [6] mit BJ- bzw. DSC-Vorkonditionierung anhand eines typischen, kleinen *linearTRACE*-Matrix-Problems (Matrixordnung: 56240; Nichtnullelemente: 2572040; Kondition: $8,4 \cdot 10^6$). Die Performance-Untersuchung wurde auf 1 bis 64 Prozessoren eines Cluster von DLR-AT durchgeführt (AMD Opteron Processor 250; Dual-Processor Nodes; 2,4 GHz).

Abbildung 3 (links) zeigt, dass bei kleinen Prozessorzahlen das unaufwendigste BJ-Verfahren (unvollständige *LU*-Zerlegung der Diagonalblöcke mit Threshold 10^{-2} [6]) die günstigsten Ausführungszeiten liefert. Ab 16 Prozessoren wird das etwas teuere BJ-Verfahren mit Threshold 10^{-3} gleichwertig und erzielt ab 32 Prozessoren ein besseres Zeitverhalten. Ab 32 Prozessoren lohnt sich die aufwendigere DSC-Methode mit dem besten Parametersatz (Threshold 10^{-3}, 5 innere Iterationen für das Kopplungssystem [8]) und liefert ab 64 Prozessoren ca. 25% kürzere Ausführungszeiten als die beste BJ-Methode (Abbildung 3, rechts).

Die Performance-Untersuchung lässt den Schluss zu, dass sich die aufwendige DSC-Methode gegenüber dem einfachen BJ-Verfahren bei hohen Prozessorzahlen für *linearTRACE*-Probleme lohnt. Wünschenswert ist ein intelligenter Löser mit Problem- bzw. konvergenzabhängiger Parametersteuerung und Problem- bzw. konvergenzabhängiger Vorkonditionierung (z.B. BJ bei kleinen Prozessorzahlen und DSC bei hohen Prozessorzahlen).

Für den auf unstrukturierten Netzen arbeitenden Strömungslöser TAU soll ferner untersucht werden, ob sich die parallele Effizienz des verwendeten Mehrgitterverfahrens weiter erhöhen lässt.

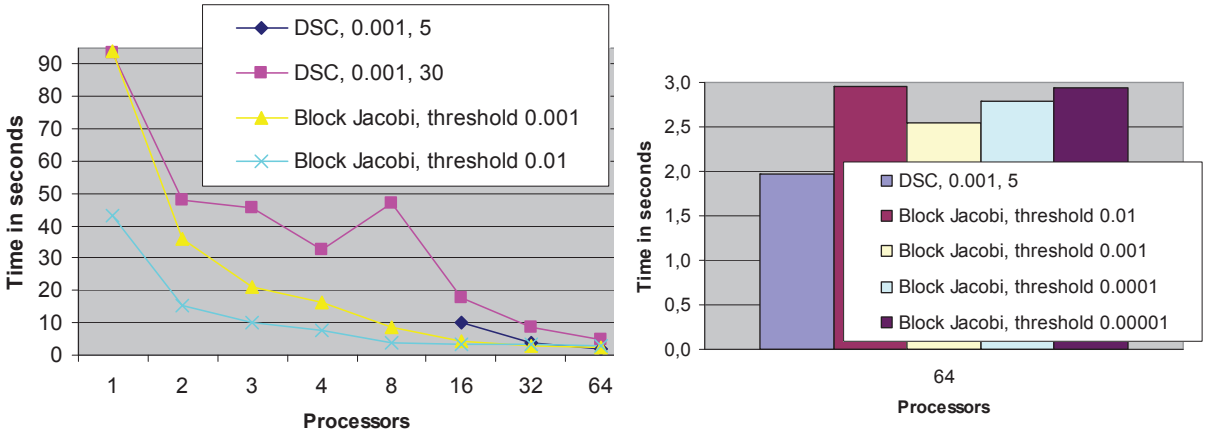

Abb. 3. Performance-Vergleich eines parallelen FGMRes-Lösers mit BJ- bzw. DSC-Vorkonditionierung.

4 Hybride OpenMP/MPI-Implementierung TAU/TRACE

Beim TAU-Strömungslöser wird eine mehrstufige SPMD-Gebietszerlegung implementiert (SPMD/MPI – SPMD/OpenMP). Die MPI-Domänen selbst werden dabei vollständig in Sub-Domänen zerlegt. Diese Sub-Domänen sind dabei auf die Größen der entsprechenden L2- bzw. L3-Caches angepasst. Die entstehende hohe zeitliche und räumliche Datenlokalität erlaubt eine hocheffiziente Nutzung der Caches. Mit Hilfe der SPMD/OpenMP-Implementierung können die Sub-Domänen parallel abgearbeitet werden. Durch die zweistufige MPI/OpenMP-Parallelisierung kann eine extrem hohe parallele Effzienz erreicht werden. Insbesondere ist der Synchronisationsaufwand zwischen den OpenMP-Threads auf den Cores relativ gering. Zusätzlich muss in diesem Schritt auch das Zusammenspiel zwischen dem SIMD-Präprozessor und den numerischen Kernels von TAU gewährleistet werden.

Beim TRACE-Strömungslöser kommt ebenfalls eine hybride OpenMP/MPI-Implementierung zum Tragen. Durch die von TRACE verwendeten blockstrukturierten Gitter ist eine klassische, schleifenbasierte OpenMP-Lösung pro MPI-Domäne erfolgversprechend. Hierbei soll ein Analysewerkzeug auf Schleifenebene geschaffen werden, das die notwendigen Direktiven unter Berücksichtigung von shared und private Variablen automatisch einfügt und die Schleifen restrukturieren und optimieren kann. Somit steht dem Modellierer bei der Erweiterung des Simulationsprogramms die hybride Parallelisierung direkt zur Verfügung.

In Analogie zum TAU-Strömungslöser muss hier ebenfalls das Zusammenspiel zwischen dem SIMD-Präprozessor und den numerischen Kernels von TRACE gewährleistet werden.

Basierend auf der Erfahrung beim Umgang mit den numerischen Kernels der beiden Strömungslöser werden beide Applikationen mit entsprechenden Direktiven instrumentiert.

5 Generischer SIMD-Präprozessor

Basierend auf den Anforderungen der extrahierten numerischen Kerne der beiden Strömungslöser TAU und TRACE soll ein generischer SIMD-Präprozessor entwickelt werden.

Das Ziel ist dabei die Entwicklung eines Programmierwerkzeuges, das einerseits automatisch den Quellcode eines Programms transformiert, andererseits dem Benutzer über eine grafische Oberfläche erlaubt, gezielt selbst Quellcodebereiche zur Transformation auszuwählen (siehe Abb. 4). Damit wird dem Entwickler sowohl bei der SIMD-Optimierung als auch bei der Optimierung des Cache-Verhaltens die fehleranfällige und langwierige Arbeit der Code-Änderung abgenommen.

Durch Direktiven soll der Entwickler Quellcodetransformationen anweisen können, die der Präprozessor umsetzt und den so generierten neuen Quellcode dem Entwickler zur Kontrolle präsentiert. Weiterhin sollen auf Anweisung hin automatisch Instrumentierungsroutinen für die Vampir-Suite in den Quellcode eingefügt werden, um mit Hilfe der erweiterten Tracing-Möglichkeiten der Vampir-Suite (siehe Abschnitt 2) entstandene Änderungen überprüfen zu können. Die Implementierung des SIMD-Präprozessors soll auf Vorarbeiten eines Präprozessors zur Cache-Optimierung aufsetzen, welcher im Rahmen eines Forschungsprojektes an der TU Dresden entwickelt wurde [9].

Der Präprozessor soll in zwei Stufen implementiert werden. In der ersten Stufe werden die folgenden allgemeinen Code- und Schleifentransformationen implementiert:

1. *Inlining:* (Beim Inlining wird innerhalb des Quellcodes der Aufruf einer Funktion durch den Funktionskörper der aufrufenden Funktion explizit ersetzt.) Damit überhaupt weitere Quellcodetransformationen möglich sind, sollen Funktionsaufrufe in der innersten Schleife mittels Inlining ersetzt werden. Der Programmierer muss dabei dafür sorgen, dass die Parameterliste keine Alias-Argumente enthält.
2. *Schleifeninvariante/indexvariante Verzweigung:* (Eine Verzweigung in einer Schleife, bei der das Ergebnis von Daten des aktuellen Schleifendurchlaufs unabhängig ist, heißt schleifeninvariante Verzweigung, eine Verzweigung innerhalb einer Schleife, bei der das Ergebnis nur vom Schleifenindex abhängig ist, heißt indexvariante Verzweigung). Bei einer schleifeninvarianten Verzweigung soll der Schleifenkörper zerlegt werden, bei einer indexvarianten Verzweigung sollen beide Möglichkeiten gerechnet und danach ein Ergebnis verworfen werden.
3. *Loop-Unrolling:* (Ist eine Schleife mit einer festen Anzahl m an Schleifendurchläufen vorgegeben, dann wird die Schleife *abgerollt,* falls die Anweisungen im Schleifenkörper mit entsprechend angepassten Schleifenindizes n-mal ($n \leq m$) explizit wiederholt werden [10, S. 180]).
4. *Loop-Splitting:* (Ist eine Schleife gegeben, die in ihrem Schleifenkörper mehrere, voneinander unabhängige Anweisungen besitzt, dann wird die Schleife *aufgeteilt,* falls für jede Anweisung im Schleifenkörper eine eigene, neue Schleife erstellt wird [11, S. 666].) Beim Loop-Splitting sollen skalare Variablen in so genannte Workarrays überführt werden. Dabei ist zu beachten, dass diese Arrays von der Datengröße her in den L1-Cache passen. Dies lässt sich typischerweise mittels Loop-Blocking der äußeren Schleife und entsprechenden inneren Schleifen mit Längen von maximal 8 bis 16 erreichen.

Die obigen Transformationen bilden die Grundlage für die zweite Stufe. In dieser Stufe sollen dann Quellcodeteile entweder direkt durch so genannte *intrinsische SIMD-Funktionen* (spezielle Funktionen eines Compilers, die SIMD-Maschinenbefehle verwenden) ersetzt werden oder der Compiler bei ihrer automatischen Vektorisierung durch den Präprozessor unterstützt werden. Anhand von verschiedenen Benchmarks soll entschieden werden, welcher Ansatz gewählt wird.

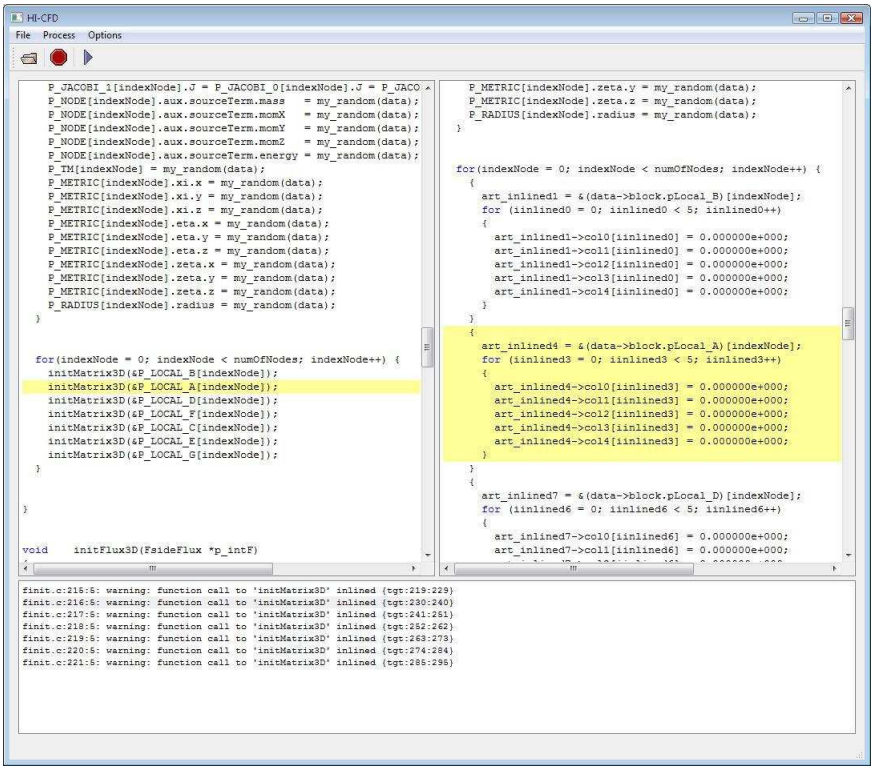

Abb. 4. Schnappschuss der grafischen Oberfläche des SIMD-Präprozessors.

6 Evaluierung TAU/TRACE

Der erfolgreiche Einsatz der Strömungslöser in Forschung und Industrie hängt nachhaltig vom Qualitätsmanagement der Software ab. Hierbei seien die Aspekte Konsistenz der Lösungen zu früheren Versionen, numerische Stabilität als auch Wartbarkeit für die vielfältige weitere Entwicklung auf verschiedenen Modellebenen genannt. Dies ist Voraussetzung, damit die in diesem Projekt geänderten Strömungslöser eine Basis für alle zukünftigen Entwicklungen bilden können.

Eine Evaluierung an industrierelevanten Testfällen soll die erhöhte Wettbewerbsfähigkeit der in HICFD modifizierten Simulationscodes demonstrieren.

Es wird ein Best-Practice-Report erstellt, der die Themengebiete „Hybride Parallelisierung“ und „SIMD-Nutzung“ behandeln wird. Hierin werden Vorgehensweisen, Hinweise und Erfahrungen aus diesem Projekt dem wissenschaftlichen Umfeld zur Verfügung gestellt. Der Best-Practice-Report soll ferner als Anleitung dienen, wie Simulationscodes auch aus anderen Disziplinen als der Strömungsmechanik für die effiziente Ausführung auf Many-Core-Architekturen optimiert werden können.

Danksagung. Das Projekt wird vom Bundesministerium für Bildung und Forschung unter den Kennzeichen IH08012A, B, C im Rahmen des Programms „IKT 2020 – Forschung und Innovation“ gefördert.

Literatur

1. Hartmut Mix und Anika Uhlemann: VampirServer 1.09 User Manual. GWT-TUD, Dresden. (2008)
2. B. Sprunt: The Basics of Performance-monitoring Hardware. IEEE Micro **22**(4) (2002) 64–71

3. Dan Terpstra: PAPI Users Guide Version 3.5.0. Innovative Computing Laboratory, University of Tennessee Department of Electrical Engineering and Computer Science, Knoxville. (2006)
4. TU Dresden, ZIH: VampirTrace 5.6 User Manual. TU Dresden, ZIH, Dresden. (2008)
5. G. Karypis und V. Kumar: ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library. University of Minnesota, Minneapolis. (1997)
6. Yousef Saad: Iterative Methods for Sparse Linear Systems (Second Edition). SIAM, Philadelphia (2003)
7. Y. Saad und M. Sosonkina: Distributed Schur Complement Techniques for General Sparse Linear Systems. SISC **21** (1999) 1337–1356
8. A. Basermann, U. Jaekel, M. Nordhausen und K. Hachiya: Parallel Iterative Solvers for Sparse Linear Systems in Circuit Simulation. Future Generation Computer Systems **21** (2005) 1275–1284
9. R. Müller-Pfefferkorn, W. E. Nagel und B. Trenkler: Optimizing Cache Access: A Tool for Source-To-Source Transformations and Real-Life Compiler Tests. M. Danelutto, M. Vanneschi, D. Laforenza (Eds): Euro-Par 2004, Parallel Processing **LNCS 3149** (2004) 72–81
10. Wolfram Schiffmann und Robert Schmitz: Technische Informatik 2: Grundlagen der Computertechnik. Springer-Verlag (2005)
11. Randy Allen und Ken Kennedy: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. NetLibrary (2001)